Well-typed Islands Parse Faster

Erik Silkensen
University of Colorado at Boulder
erik.silkensen@colorado.edu

Jeremy Siek
University of Colorado at Boulder
jeremy.siek@colorado.edu

Abstract

This paper addresses the problem of specifying and parsing the syntax of domain-specific languages (DSLs) in a modular, user-friendly way. That is, we want to enable the design of *composable* DSLs that combine the natural syntax of external DSLs with the easy implementation of internal DSLs. The challenge in parsing composable DSLs is that the composition of several (individually unambiguous) languages is likely to contain ambiguities. In this paper, we present the design of a system that uses a type-oriented variant of island parsing to efficiently parse the syntax of composable DSLs. In particular, we show how type-oriented island parsing is constant time with respect to the number of DSLs imported. We also show how to use our tool to implement DSLs on top of a host language such as Typed Racket.

1. Introduction

Domain-specific languages (DSLs) provide high productivity for programmers in many domains, such as computer systems, physics, linear algebra, and other sciences. However, a series of trade-offs face the prospective DSL designer today. On the one hand, external DSLs offer natural syntax and friendly diagnostics at the cost of interoperability issues [Beazley 1996] and difficulty of implementation. They are usually either implemented by hand or by using parser generators *à la* YACC that require technical knowledge of parsing algorithms. Meanwhile, many general-purpose languages include a host of tricks for implementing internal (or embedded) DSLs, such as templates in C++, macros in Scheme, and type classes in Haskell; however, the resulting DSLs are often *leaky abstractions*: the syntax is not quite right, compilation errors expose the internals of the DSL, and debuggers are not aware of the DSL.

In this paper, we make progress towards combining the best of both worlds into what we call *composable* DSLs. We want to enable fine-grained mixing of languages with the natural syntax of external DSLs and the interoperability of internal DSLs.

At the core of this effort is a parsing problem: although the grammar for each DSL may be unambiguous, programs that use multiple DSLs, such as the one in Figure 1, need to be parsed using the union of their grammars, which are likely to contain ambiguities [Kats et al. 2010]. Instead of relying on the grammar author to resolve them (as in the LALR tradition), the parser for such an application must be able to efficiently deal with ambiguities.

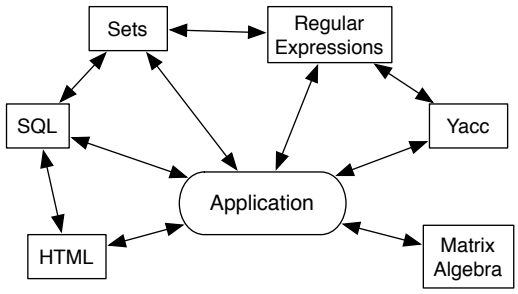


Figure 1. Our common case: an application using many DSLs.

We should emphasize that our goal is to create a parsing system that provides much more syntactic flexibility than is currently offered through operator overloading in languages such as C++ and Haskell. We are not trying to build a general purpose parser, that is, we are willing to place restrictions on the allowable grammars, so long as those restrictions are easy to understand (for our users) and do not interfere with composability.

As a concrete, motivating example, we consider the union of grammars for matrix algebra, regular expressions, and sets outlined in Figure 2. Written in the traditional style, the union of these individually unambiguous grammars is greatly ambiguous; so importing many DSLs such as these can increase the parse time by orders of magnitude even though the program is otherwise unchanged. Of course, an experienced computer scientist will immediately say that the separate grammars should be merged into one grammar with only one production for each operator. However, that would require coordination between the DSL authors and is therefore not scalable.

1.1 Type-Oriented Grammars

To address the problem of parsing composed DSLs, we observe that different DSLs define different types: *Matrix*, *Vector*, and *Scalar* in Matrix Algebra, *Regexp* in Regular Expressions, *Set* in Sets, and so on. We suggest an alternate style of grammar organization that we call *type-oriented grammars*, inspired by Sandberg [1982]. In this style, a DSL author creates one nonterminal for each type in the DSL and uses the most specific nonterminal/type for each operand in a grammar rule. Figure 3 shows the example from Figure 2 rewritten in a type-oriented style, with nonterminals for `Matrix`, `Vector`, `Scalar`, `Regexp`, and `Set`.

1.2 Type-based Disambiguation

While the union of the DSLs in Figure 3 is no longer itself ambiguous, programs such as $\texttt{A} + \texttt{B} + \texttt{C} \cdots$ are still highly ambiguous if the variables `A`, `B`, and `C` can each be parsed as either `Matrix`, `Regexp`, or `Set`. Many prior systems [Paulson 1994, Bravenboer et al. 2005] use chart parsing [Kay 1986] or GLR [Tomita 1985]

```
module MatrixAlgebra {
  Expr ::= Expr "+" Expr [left,1]
         | Expr "-" Expr [left,1]
         | Expr "*" Expr [left,2]
         | "|" Expr "|" | Id; ···
}
module RegularExpressions {
  Expr ::= "'" Char "'" | Expr "+" | Expr "*"
         | Expr "|" Expr [left] | Id; ···
}
module Sets {
  Expr ::= Expr "+" Expr [left,1]
         | Expr "-" Expr [left,2] | Id; ···
}

import MatrixAlgebra, RegularExpressions, Sets;
A + B + C      // Ambiguous!
```

Figure 2. Ambiguity due to the union of DSLs.

```
module MatrixAlgebra {
  Matrix ::= Matrix "+" Matrix [left,1]
           | Matrix "-" Matrix [left,1]
           | Matrix "*" Matrix [left,2];
  Scalar ::= "|" Vector "|"; ···
}
module RegularExpressions {
  Regexp ::= "'" Char "'" | Regexp "+"
           | Regexp "*" | Regexp "|" Regexp; ···
}
module Sets {
  Set ::= Set "+" Set [left,1]
        | Set "-" Set [left,2]; ···
}

import MatrixAlgebra, RegularExpressions, Sets;
declare A:Matrix, B:Matrix, C:Matrix {
  A + B + C
}
```

Figure 3. Type-oriented grammars for DSLs.

to produce a parse forest and then type check to filter out the ill-typed trees. This solves the ambiguity problem, but these parsers are inefficient on ambiguous grammars (Section 4).

This is where our key contribution comes in: *island parsing with eager, type-based disambiguation*. We use a chart parsing strategy, called island parsing [Stock et al. 1988] (or bidirectional bottom-up parsing [Quesada 1998]), that enables our algorithm to grow parse trees outwards from *well-typed terminals*. The statement

declare `A:Matrix, B:Matrix, C:Matrix` { ... }

gives the variables `A`, `B`, and `C` the type `Matrix`. We then integrate type checking into the parsing process to prune ill-typed parse trees before they have a chance to grow, drawing inspiration from from the field of natural language processing, where using types to resolve ambiguity is known as *selection restriction* [Jurafsky and Martin 2009],

Our approach does not altogether prohibit grammar ambiguities; it strives to remove ambiguities from the common case when composing DSLs so as to enable efficient parsing.

1.3 Contributions

1. We present the first parsing algorithm, *type-oriented island parsing* (Section 3), whose time complexity is *constant* with respect to the number of DSLs in use, so long as the nonterminals of each DSL are largely disjoint (Section 4).
2. We present our extensible parsing system[1] that adds several features to the parsing algorithm to make it convenient to develop DSLs on top of a host language such as Typed Racket [Tobin-Hochstadt and Felleisen 2008] (Section 5).
3. We demonstrate the utility of our parsing system with an example in which we embed syntax for two DSLs in Typed Racket.

Section 2 introduces the basic definitions and notation used in the rest of the paper. We discuss our contributions in relation to the prior literature in Section 6 and conclude in Section 7.

2. Background

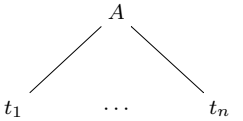
We review the definition of a grammar and parse tree and present our framework for comparing parsing algorithms, which is based on the parsing schemata of Sikkel [1998].

2.1 Grammars and Parse Trees

A *context-free grammar* (CFG) is a 4-tuple $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ where Σ is a finite set of terminals, Δ is a finite set of nonterminals, $\mathcal{P}$ is finite set of grammar rules, and S is the start symbol. We use a, b, c, and d to range over terminals and A, B, C, and D to range over nonterminals. The variables X, Y, Z range over symbols, that is, terminals and nonterminals, and $\alpha, \beta, \gamma, \delta$ range over sequences of symbols. Grammar rules have the form $A \to \alpha$. We write $\mathcal{G} \cup (A \to \alpha)$ as an abbreviation for $(\Sigma, \Delta, \mathcal{P} \cup (A \to \alpha), S)$.

We are ultimately interested in parsing programs, that is, converting token sequences into abstract syntax trees. So we are less concerned with the recognition problem and more concerned with determining the parse trees for a given grammar and token sequence. The *parse trees* for a grammar $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$, written $\mathcal{T}(\mathcal{G})$, are trees built according to the following rules.

1. If $a \in \Sigma$, then a is a parse tree labeled with a.
2. If $t_1, \ldots, t_n$ are parse trees labeled $X_1, \ldots, X_n$ respectively, $A \in \Delta$, and $A \to X_1, \ldots, X_n \in \mathcal{P}$, then the following is a parse tree labeled with A.



We sometimes use a horizontal notation $A \to t_1 \ldots t_n$ for parse trees and we often subscript parse trees with their labels, so t_A is parse tree t whose root is labeled with A. We use an overline to represent a sequence: $\overline{t} = t_1, \ldots, t_n$.

The *yield* of a parse tree is the concatenation of the labels on its leaves:

$$yield(a) = a$$
$$yield([A \to t_1 \ldots t_n]) = yield(t_1) \ldots yield(t_n)$$

Definition 2.1. The set of parse trees for a CFG $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ and input w, written $\mathcal{T}(\mathcal{G}, w)$, is defined as follows

$$\mathcal{T}(\mathcal{G}, w) = \{t_S \mid t_S \in \mathcal{T}(\mathcal{G}) \text{ and } yield(t_S) = w\}$$

Definition 2.2. The *language* of a CFG $\mathcal{G}$, written $L(\mathcal{G})$, consists of all the strings for which there is exactly one parse tree. More

[1] See the supplemental material for the URL for the code.

formally,

$$L(\mathcal{G}) = \{w \mid |\mathcal{T}(\mathcal{G}, w)| = 1\}$$

2.2 Parsing Algorithms

We wish to compare the essential characteristics of several parsing algorithms without getting distracted by implementation details. Sikkel [1998] introduces a high-level formalism for presenting and comparing parsing algorithms, called *parsing schemata*, that present each algorithm as a deductive system. We loosely follow his approach, but make some changes to better suit our needs.

Each parsing algorithm corresponds to a deductive system with judgments of the form

$$H \vdash \xi$$

where ξ is an *item* and H is a set of items. An item has the form $[p, i, j]$ where p is either a parse tree or a partial parse tree and the integers i and j mark the left and right extents of what has been parsed so far. The set of *partial parse trees* is defined by the following rule.

> If $A \to \alpha\beta\gamma \in \mathcal{P}$, then $A \to \alpha.\bar{t}_\beta.\gamma$ is a partial parse tree labeled with A.

We reserve the variables s and t for parse trees, not partial parse trees. A *complete parse* of an input w of length n is a derivation of $H_0 \vdash [t_S, 0, n]$, where H_0 is the initial set of items that represent the result of tokenizing the input w.

$$H_0 = \{[w_i, i, i+1] \mid 0 \le i < |w|\}$$

Example 2.3. The (top-down) Earley algorithm [Earley 1968, 1970] applied to a grammar $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ is defined by the following deductive rules.

$$\text{(Hyp)}\frac{\xi \in H}{H \vdash \xi} \qquad \text{(Fnsh)}\frac{H \vdash [A \to .\bar{t}_\alpha., i, j]}{H \vdash [A \to \bar{t}_\alpha, i, j]}$$

$$\text{(Init)}\frac{S \to \gamma \in \mathcal{P}}{H \vdash [S \to ..\gamma, 0, 0]}$$

$$\text{(Pred)}\frac{H \vdash [A \to .\bar{t}_\alpha.B\beta, i, j] \qquad B \to \gamma \in \mathcal{P}}{H \vdash [B \to ..\gamma, j, j]}$$

$$\text{(Compl)}\frac{H \vdash [A \to .\bar{s}_\alpha.X\beta, i, j] \qquad H \vdash [t_X, j, k]}{H \vdash [A \to .\bar{s}_\alpha t_X.\beta, i, k]\}}$$

Example 2.4. A bottom-up variation [Sikkel 1998] of Earley parsing is obtained by replacing the initialization (Init) and prediction (Pred) rules with the following bottom-up rule (BU).

$$\text{(BU)}\frac{H \vdash [t_X, i, j] \qquad A \to X\beta \in \mathcal{P}}{H \vdash [A \to .t_X.\beta, i, j]}$$

3. Type-Oriented Island Parsing

The essential ingredients of our parsing algorithm are type-based disambiguation and island parsing. In Section 4. we show that an algorithm based on these two ideas parses with time complexity that is independent of the number of DSLs in use, so long as the nonterminals of the DSLs are largely disjoint. (We also make this claim more precise.) But first, in this section we introduce our type-oriented island parsing algorithm (TIP).

Island parsing [Stock et al. 1988] is a bidirectional, bottom-up parsing algorithm that was developed in the context of speech recognition. In that domain, some tokens can be identified with a higher confidence than others. The idea of island parsing is to begin the parsing process at the high confidence tokens, the so-called islands, and expand the parse trees outward from there.

Our main insight is that if our parser can be made aware of variable declarations, and if a variable's type corresponds to a nonterminal in the grammar, then each occurrence of a variable is treated as an island. We introduce the following special form for declaring a variable a of type A that may be referred to inside the curly brackets.

$$\textbf{declare}\ a : A\ \{\ldots\}$$

For the purposes of parsing, the rule $A \to a$ is added to the grammar while parsing inside the curly brackets. To enable temporarily extending the grammar, we augment the judgments of our deductive system with an explicit parameter for the grammar. So judgments have the form

$$\mathcal{G}; H \vdash \xi$$

This adjustment also enables the import of grammars from different modules.

We formalize the parsing rule for the **declare** form as follows.

$$\text{(Decl)}\frac{\mathcal{G} \cup (A \to a); H \vdash [t_X, i+5, j]}{\mathcal{G}; H \vdash [X \to \textbf{declare}\ a : A\ \{t_X\}, i, j+1]}$$

Next we split the bottom-up rule (BU) into the two following rules. The (Island) rule triggers the formation of an island using grammar rules of the form $A \to a$, which arise from variable declarations and from literals (constants) defined in a DSL. The (IPred) rule generates items from grammar rules that have the parsed nonterminal B on the right-hand side.

$$\text{(Island)}\frac{\mathcal{G}; H \vdash [a, i, j] \qquad A \to a \in \mathcal{P} \qquad \mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)}{\mathcal{G}; H \vdash [A \to a, i, j]}$$

$$\text{(IPred)}\frac{\mathcal{G}; H \vdash [t_B, i, j] \qquad A \to \alpha B\beta \in \mathcal{P} \qquad \mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)}{\mathcal{G}; H \vdash [A \to \alpha.t_B.\beta, i, j]}$$

Finally, because islands appear in the middle of the input string, we need both left and right-facing versions of the (Compl) rule.

$$\text{(RCompl)}\frac{\mathcal{G}; H \vdash [A \to \alpha.\bar{s}_\beta.X\gamma, i, j] \qquad \mathcal{G}; H \vdash [t_X, j, k]}{\mathcal{G}; H \vdash [A \to \alpha.\bar{s}_\beta t_X.\gamma, i, k]\}}$$

$$\text{(LCompl)}\frac{\mathcal{G}; H \vdash [t_X, i, j] \qquad \mathcal{G}; H \vdash [A \to \alpha X.\bar{s}_\beta.\gamma, j, k]}{\mathcal{G}; H \vdash [A \to \alpha.t_X\bar{s}_\beta.\gamma, i, k]\}}$$

Definition 3.1. The *type-oriented island parsing* algorithm is defined as the deductive system comprised of the rules (Hyp), (Fnsh), (Decl), (Island), (IPred) (RCompl), and (LCompl).

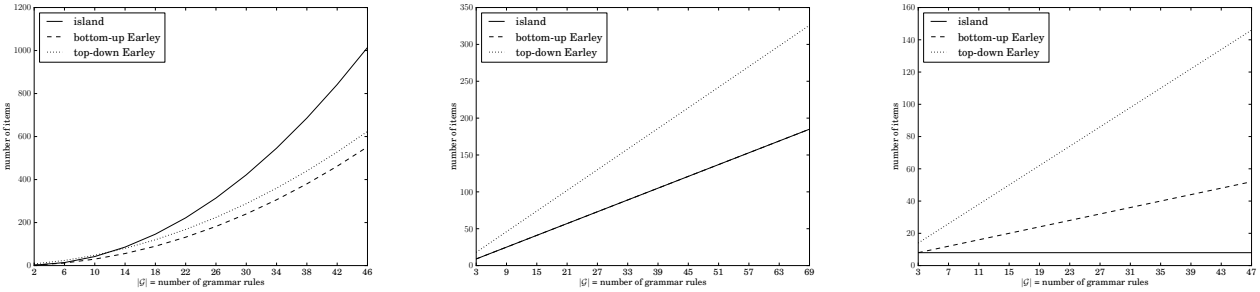
The type-oriented island parsing algorithm requires a minor restriction on grammars. If the right-hand side of a rule does not contain any nonterminals, then it may only contain a single terminal. This restriction means that our system supports single-token literals but not multi-token literals. For example, the grammar rule $A \to$ `"foo" "bar"` is not allowed, but $A \to$ `"foobar"` and $A \to$ `"foo"` B `"bar" "baz"` are allowed.

4. Experimental Evaluation

In this section we evaluate the performance of type-oriented island parsing with experiments in two separate dimensions. First we measure the performance of the algorithm for programs that are held constant but the size of the grammars increase, and second we measure the performance for programs that increase in size while the grammars are held constant.

4.1 Grammar Scaling

Chart parsing algorithms [Kay 1986] have a general worst-case running time of $O(|\mathcal{G}|n^3)$ for a grammar $\mathcal{G}$ and string of length n.



(a) Untyped Grammar (b) Semi-typed Grammar (c) Type-oriented Grammar

Figure 4. Comparison of top-down, bottom-up, and island parsing with three styles of grammars.

In our setting, $\mathcal{G}$ is the union of the grammars for all the k DSLs that are in use within a given scope, that is $\mathcal{G} = \bigcup_{i=1}^{k} \mathcal{G}_i$, where $\mathcal{G}_i$ is the grammar for DSL i. We claim that the total size of the grammar $\mathcal{G}$ is not a factor for type-oriented island parsing, and instead the time complexity is $O(mn^3)$ where $m = \max\{|\mathcal{G}_i| \mid 1 \leq i \leq k\}$. This claim deserves considerable explanation to be made precise.

Technically, we assume that $\mathcal{G}$ is *sparse* and that the terminals of $\mathcal{G}$ are *well-typed*, which we define as follows.

Definition 4.1. Form a Boolean matrix with a row for each nonterminal and a column for each production rule in a grammar $\mathcal{G}$. A matrix element (i, j) is `true` if the nonterminal i appears on the right-hand side of the rule j, and it is `false` otherwise. We say that $\mathcal{G}$ is *sparse* if its corresponding matrix is sparse, that is, if the number of nonzero elements is much smaller than the number of elements.

Definition 4.2. We say that a terminal a of a grammar $\mathcal{G}$ is *well-typed* if for each B such that $B \to a \in \mathcal{P}$, B represents a type in the language of $\mathcal{G}$.

We expect that the terminals of type-oriented grammars will be well-typed, and hypothesize that, in the common case, the union of many type-oriented grammars (or DSLs) is sparse.

To verify that both the type-oriented style of grammar and the island parsing algorithm are necessary for this result, we show that removing either of these ingredients results in parse times that are dependent on the size of the entire grammar. Specifically, we consider the performance of the top-down and bottom-up Earley algorithms, in addition to island parsing, with respect to untyped, semi-typed, and type-oriented grammars.

We implemented all three algorithms in a chart parsing framework [Kay 1986], which efficiently memoizes duplicate items. The chart parser continues until it has generated all items that can be derived from the input string. (It does not stop at the first complete parse because it needs to continue to check whether the input string is ambiguous, which means the input would be in error.) Also, we should note that our system currently employs a fixed tokenizer, but that we plan to look into scannerless parsing.

To capture the essential, asymptotic behavior of the parsing algorithms, we measure the number of items generated during the parsing of the program.

4.1.1 A Small Experiment

For the first experiment we parse the expression `--A` with untyped, semi-typed, and typed grammars.

Untyped In the untyped scenario, all grammar rules are defined in terms of the expression nonterminal (`E`), and variables are simply parsed as identifiers (`Id`).

```
module Untyped^k {
  E ::= Id | "-" E;
}
```

The results for parsing `--A` after importing k copies of `Untyped`, for increasing k, are shown in Figure 4(a). The y-axis is the number of items generated by each parsing algorithm, and the x-axis is the total number of grammar rules at each k. In the untyped scenario, the size of the grammar affects the performance of each algorithm, with each generating $O(k^2)$ items.

We note that the two Earley algorithms generate about half as many items as the island parser because they are unidirectional (left-to-right) instead of bidirectional.

Semi-typed In the semi-typed scenario, the grammars are nearly type-oriented: the `Semityped`^{0} module defines the nonterminal `V` (for vector) and each of `Semityped`^{i} for $i \in \{1, \ldots, k\}$ defines the nonterminal `M`i (for matrix); however, variables are again parsed as identifiers. We call this scenario *semi-typed*, because it doesn't use variable declarations to provide type-based disambiguation.

```
module Semityped^0 {
  E ::= V;
  V ::= Id | "-" V;
}
module Semityped^i {
  E ::= Mi;
  Mi ::= Id | "-" Mi;
}
```

The results for parsing `--A` after importing `Semityped`^{0} followed by `Semityped`^{i} for $i \in \{1, \ldots, k\}$ are shown in Figure 4(b). The lines for bottom-up Earley and island parsing coincide. Each algorithm generates $O(k)$ items, but we see that type-oriented grammars are not, by themselves, enough to achieve constant scaling with respect to grammar size.

We note that the top-down Earley algorithm generates almost twice as many items as the bottom-up algorithms: the alternatives for the start symbol `E` grow with n, which affects the top-down strategy more than bottom-up.

Typed The typed scenario is identical to semi-typed except that it no longer includes the `Id` nonterminal. Instead, programs using the `Typed` module must declare their own typed variables.

```
module Typed^0 {
  E ::= V;
  V ::= "-" V;
}
```

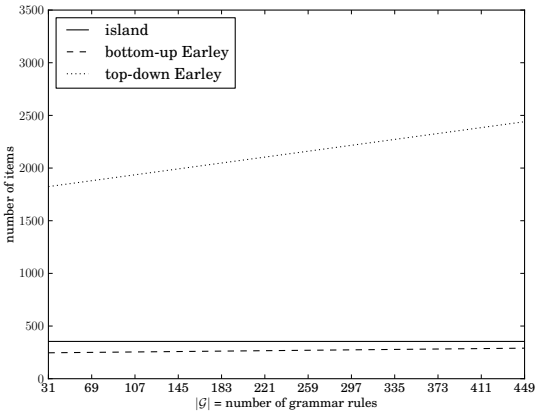


Figure 5. Comparison of parsing algorithms for a type-oriented matrix algebra DSL and increasing grammar size.

```
module Typed^i {
  E ::= M^i;
  M^i ::= "-" M^i;
}
```

The results for parsing `--A` after importing $\texttt{Typed}^0$ followed by $\texttt{Typed}^i$ for $i \in \{1, \ldots, k\}$ and declaring `A:V` are shown in Figure 4(c). The sparsity for this example is $O(1/k)$, and now the terminal (as `V`) is well-typed. The island parsing algorithm generates a constant number of items as the size of the grammar increases, while the Earley algorithms remain linear. Thus, the *combination* of type-based disambiguation, type-oriented grammars, and island parsing provides a scalable approach to parsing programs that use many DSLs.

4.1.2 A Larger Experiment

For a second experiment we measure the performance of each algorithm for a sequence of matrix algebra operations with expanded versions of the grammars in Figure 3:

```
import MatrixAlgebra, RegularExpressions^k, Sets^k;
B = A + u1 * v1' + u2 * v2';
x = b * (B' * y) + z;
w = a * (B * x);
```

In this example, we import grammars for `RegularExpressions` and `Sets`, k times each. For the untyped and semi-typed scenarios, the result is too ambiguous and we terminated their execution after waiting for several minutes. For the typed scenario, we declare the variables `A` and `B` as type `Matrix`; `u1`, `u2`, `v1`, `v2`, and `w-z` as type `ColVector`; `a` and `b` as type `Scalar`; the sparsity of the typed example is $O(1/k)$.

Figure 5 shows a graph for parsing the above program with each algorithm. As before, the y-axis is the number of items generated during parsing, and the x-axis is the number of DSLs that are imported. The top-down Earley algorithm scales linearly with respect to the number of DSLs imported and generates many more items than the bottom-up algorithms. The island parsing algorithm generates a constant number of items as the number of DSLs increases; the bottom-up Earley algorithm generates a similar number of items, but it scales slightly linearly.

4.1.3 Discussion

The reason type-oriented island parsing scales is that it is more conservative with respect to prediction than either top-down or bottom up, and so grammar rules from other DSLs that are irrelevant to the program fragment being parsed are never used to generate items.

Consider the (PRED) rule of top-down Earley parsing. Any rule that produces the non-terminal B, regardless of which DSL it resides in, will be entered into the chart. Note that such items have a zero-length extent which indicates that the algorithm does not yet have a reason to believe that this item will be able to complete.

Looking at the (BU) rule of bottom-up Earley parsing, we see that all it takes for a rule (from any DSL) to be used is that it starts with a terminal that occurs in the program. However, it is quite likely that different DSLs will have rules with some terminals in common. Thus, the bottom-up algorithm also introduces items from irrelevant DSLs.

Next, consider the (ISLND) rule of our island parser. There is no prediction in this rule. However, it is possible for different DSLs to define literals with the same syntax (same tokens). (Many languages forbid the overloading of constants, but it is allowed, for example, in Haskell.) The performance of the island parser would degrade in such a scenario, although the programmer could regain performance by redefining the syntax of the imported constants, in the same way that name conflicts can be avoided by the rename-on-import constructs provided by module systems.

Finally, consider the (IPRED) rule of our island parser. The difference between this rule and (BU) is that it only applies to nonterminals, not terminals. As we previously stated, we assume that the nonterminals in the different DSLs are, for the most part, disjoint. Thus, the (IPRED) rule typically generates items based on rules in the relevant DSL's grammar and not from other DSLs.

4.2 Program Scaling

In this section we measure the performance of each algorithm as the size of the program increases and the grammar is held constant. The program of size n is the addition of n matrices using the matrix algebra grammar from the previous section.

As before, we consider untyped, semi-typed, and typed scenarios. For these experiments we report parse times; we ran all of the experiments on a MacBook with a 2.16 GHz Intel Core 2 Duo processor and 2 GB of RAM.

Untyped The untyped scenario is exponentially ambiguous:

```
import MatrixAlgebra, RegularExpressions, Sets;
A + A + ··· + A
```

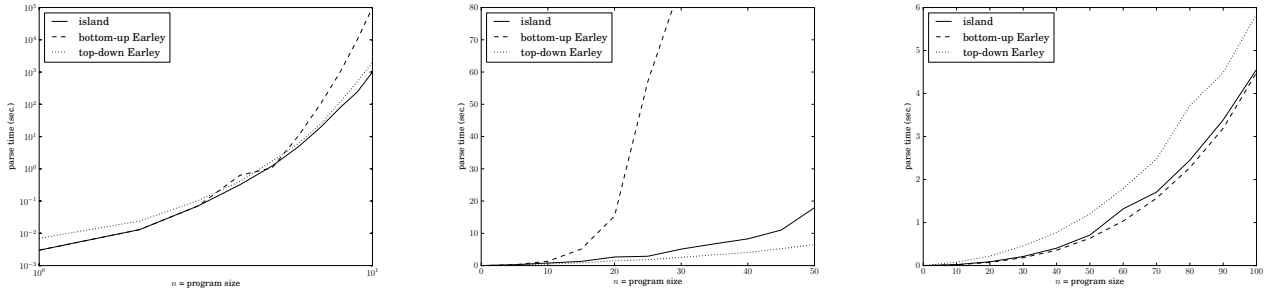
While the above program with n terms produces $O(2^n)$ parse trees, the Earley and island parsing algorithms can produce a packed parse forest in polynomial space and time [Allen 1995].

Figure 6(a) shows the results for each algorithm on a logarithmic scale. The y-axis is the parse time (including production of parse trees), and the x-axis is the program size. Because our implementation does not use packed forests, all three algorithms are exponential for the untyped scenario.

Semi-typed The program for the semi-typed scenario is identical to the untyped scenario and is also ambiguous; however, the number of parse trees doesn't grow with increasing program size. Figure 6(b) shows the results for each algorithm, now on a linear scale. The axes are the same as before. Here the top-down Earley and island algorithms are $O(n^2)$. Although the number of correct parse trees remains constant, the bottom-up Earley algorithm explores an exponential number of possible trees as n increases before returning, and uses exponential time.

Typed The program is no longer ambiguous in the typed scenario:

```
import MatrixAlgebra, RegularExpressions, Sets;
declare A:Matrix {
  A + A + ··· + A
```



(a) Untyped Grammar (b) Semi-typed Grammar (c) Type-oriented Grammar

Figure 6. Comparison of parsing algorithms for increasing program size. Figure (a) uses a logarithmic scale, with the program size ranging from 1 to 10. Figures (b) and (c) use a linear scale, with the program size ranging from 1 to 50 in (b) and ranging from 1 to 100 in (c).

}

Figure 6(c) shows the results for each algorithm on a linear scale and with axes as before. All three algorithms are $O(n^2)$ for the typed scenario. These results suggest that type-oriented Earley and island parsing are $O(n^2)$ for unambiguous grammars.

We should note that the top-down Earley algorithm parses the above program in $O(n)$ time when the grammars are rewritten to be LR(0); however, the bottom-up Earley and island algorithms remain $O(n^2)$.

5. A System for Extensible Syntax

In this section we describe the parsing system that we have built as a front end to the Racket programming language. In particular, we describe how we implement four features that are needed in a practical extensible parsing system: associativity and precedence, parameterized grammar rules, grammar rules with variable binders and scope, and rule-action pairs [Sandberg 1982] which combine of the notions of semantic actions, function definitions, and macros.

5.1 Associativity and Precedence

We view associativity and precedence annotations (as in Figure 2, e.g., `[left,1]`) as a must for our parsing system because we do not expect all of our users to be computer scientists, that is, we do not expect them to know how to manually factor a grammar to take associativity and precedence into account. Further, even for users who are computer scientists, they probably have something better to do with their time than to factor grammars.

Our treatment of associativity and precedence is largely based on that of Visser [1997], although we treat this as a semantic issue instead of an optimization issue. From the user perspective, we extend rules to have the form $A \to \alpha[\ell, p]$ where ℓ indicates the associativity, where $\ell \in \{\mathsf{left}, \mathsf{right}, \mathsf{non}, \bot\}$, and p indicates the precedence, where $p \in \mathbb{N}_\bot$. We use an ordering $<$ on precedences that is the natural lifting of $<$ on $\mathbb{N}$. (Instead of $(\mathbb{N}_\bot, <)$ we could use any partially ordered set, but prefer to be concrete here.)

To specify the semantics of precedence and associativity, we use the notion of a *filter* to remove parse trees from consideration if they contain precedence or associativity conflicts [Visser 1997]. But first, we annotate our parse trees with the precedence and associativity, so an internal node has the form $A \to_{\ell,p} \overline{t}$.

Definition 5.1. We say that a parse tree t has a *root priority conflict*, written $\mathit{conflict}(t)$, if one of the following holds.

1. It violates the right, left or non-associativity rules, that is, t has the form:
   - $A \to_{\ell,p} (A \to_{\ell,p} \overline{t}_{A\alpha})\overline{s}_\alpha$ where $\ell = \mathsf{right}$ or $\ell = \mathsf{non}$.
   - $A \to_{\ell,p} \overline{s}_\alpha(A \to_{\ell,p} \overline{t}_{\alpha A})$ where $\ell = \mathsf{left}$ or $\ell = \mathsf{non}$.
2. It violates the precedence rule, that is, t has the form:

$$t = A \to_{\ell,p} \overline{s}(B \to_{\ell',p'} \overline{t})\overline{s'} \text{ where } p' < p.$$

Definition 5.2. A *tree context* $\mathcal{C}$ is defined by the following grammar.

$$\mathcal{C} \quad ::= \quad \Box \mid A \to_{l,p} t_1 \ldots \mathcal{C} \ldots t_n$$

The operation of plugging a tree t into a tree context $\mathcal{C}$, written $\mathcal{C}[t]$, is defined as follows.

$$\Box[t] = t$$
$$(A \to_{\ell,p} t_1 \ldots \mathcal{C} \ldots t_n)[t] = A \to_{\ell,p} t_1 \ldots \mathcal{C}[t] \ldots t_n$$

Definition 5.3. The *filter* for a CFG $\mathcal{G}$ is a function on sets of trees, $\mathcal{F}\colon \wp(\mathcal{T}_\mathcal{G}) \to \wp(\mathcal{T}_\mathcal{G})$, that removes the trees containing conflicts. That is,

$$\mathcal{F}(\Phi) = \{t \in \Phi \mid \nexists t'\mathcal{C},\ t = \mathcal{C}[t'] \text{ and } \mathit{conflict}(t')\}$$

Definition 5.4. The set of parse trees for a grammar $\mathcal{G}$ (with precedence and associativity) and input w, written $\mathcal{T}(\mathcal{G}, w)$, is defined as follows.

$$\mathcal{T}(\mathcal{G}, w) = \{t_S \mid t_S \in \mathcal{F}(\mathcal{T}(\mathcal{G})) \text{ and } \mathit{yield}(t_S) = w\}$$

The change to the island parsing algorithm to handle precedence and associativity is straightforward. We simply make sure that a partial parse tree does not have a root priority conflict before converting it into a (complete) parse tree. We replace the (FNSH) rule with the following rule.

$$(\text{FNSHP})\ \frac{\mathcal{G}; H \vdash [A \to .\overline{t}_\alpha ., i, j] \qquad \neg \mathit{conflict}(A \to \overline{t}_\alpha)}{\mathcal{G}; H \vdash [A \to \overline{t}_\alpha, i, j]}$$

5.2 Parameterized Rules

With the move to type-oriented grammars, the need for parameterized rules immediately arises. For example, consider how one might translate the following grammar rule for conditional expressions into a type-oriented rule.

```
E ::= "if" E "then" E "else" E
```

We would like to be more specific than `E` for the two branches and for the left-hand side. So we extend our grammar rules to enable the parameterization of nonterminals. We can express a conditional expression as follows, where `T` stands for any type/nonterminal.

```
forall T.
  T ::= "if" Bool "then" T "else" T
```

To simplify the presentation, we describe parameterized rules as an extension to the base island parser (without precedence). However, our parsing system combines both extensions. Here we extend grammar rules to include parameters: $\forall \overline{x}.\, A \to \alpha$. ($\overline{x}$ may not contain duplicates.) We use x, y, z to range over variables and we now use the variables A, B, C, D to range over nonterminals and variables.

To handle parameters we need the notion of a substitution σ, that is, a partial function from variables to nonterminals. The initial substitution σ_0 is everywhere undefined. We extend the action of a substitution to all symbols, sequences, and rules in the following natural way.

$$\sigma(a) = a$$
$$\sigma(X_1 \ldots X_n) = \sigma(X_1) \ldots \sigma(X_n)$$
$$\sigma(A \to \alpha) = \sigma(A) \to \sigma(\alpha)$$

The notation $\sigma[X \mapsto Y]$ creates a new, extended substitution, defined as follows.

$$\sigma[X \mapsto Y](Z) = \begin{cases} Y & \text{if } X = Y, \\ \sigma(Z) & \text{otherwise.} \end{cases}$$

We write $\sigma[\overline{X} \mapsto \overline{Y}]$ to abbreviate

$$\sigma[X_1 \mapsto Y_1] \cdots [X_{|X|} \mapsto Y_{|Y|}].$$

We write $[\overline{X} \mapsto \overline{Y}]$ to abbreviate $\sigma_0[\overline{X} \mapsto \overline{Y}]$.

Next we update the definition of a parse tree to include parameterized rules. The formation rule for leaves remains unchanged, but the rule for internal nodes becomes as follows.

> If $A \in \Delta$, $\forall \overline{x}.\, A \to \alpha \in \mathcal{P}$, and $\sigma = [\overline{x} \mapsto \overline{B}]$, then $\sigma(A) \to \overline{t}_{\sigma(\alpha)}$ is a parse tree labeled with $\sigma(A)$.

The definition of the language of a CFG with parameterized rules requires some care because parameterized rules introduce ambiguity. For example, consider the parameterized rule for conditional expressions given above and the following program.

```
if true then 0 else 1
```

Instantiating parameter `T` with either `Int` or `E` leads to a complete parse. Of course, instantiating with `Int` is better in that it is more specific. We formalize this notion as follows.

Definition 5.5. We inductively define whether A *is at least as specific as* B, written $A \geq B$, as follows.

1. If $B \to A \in \mathcal{P}$, then $A \geq B$.
2. (reflexive) $A \geq A$.
3. (transitive) If $A \geq B$ and $B \geq C$, then $A \geq C$.

We extend this ordering to terminals by defining $a \geq b$ iff $a = b$, and to sequences by defining

$$\alpha \geq \beta \text{ iff } |\alpha| = |\beta| \text{ and } \alpha_i \geq \beta_i \text{ for } i \in \{1, \ldots, |\alpha|\}$$

A parse tree node $A \to \overline{s}_\alpha$ is at least as specific as another parse tree node $B \to \overline{t}_\beta$ if and only if $A \geq B$ and $\overline{s}_\alpha \geq \overline{t}_\beta$.

We define the least upper bound, $A \vee B$, with respect to the $\geq$ relation in the usual way. Note that a least upper bound does not always exist.

Definition 5.6. The *language* of a CFG $\mathcal{G}$ with parameterized rules, written $L(\mathcal{G})$, consists of all the strings for which there is a most specific parse tree. More formally,

$$L(\mathcal{G}) = \{w \mid \exists t \in \mathcal{T}(\mathcal{G}, w).\, \forall t' \in \mathcal{T}(\mathcal{G}, w).\, t' \neq t \to t \geq t'\}$$

Next we turn to augmenting our island parsing algorithm to deal with parameterized rules. We wish to implicitly instantiate parameterized grammar rules, that is, automatically determine which nonterminals to substitute in for the parameters. Towards this end, we define a partial function named $match$ that compares two symbols with respect to a substitution σ and list of variables $\overline{y}$ and produces a new substitution σ' (if the match is successful).

$$match(X, X, \sigma, \overline{y}) = \sigma$$
$$match(x, Y, \sigma, \overline{y}) = \begin{cases} \sigma[x \mapsto X \vee Y] & \text{if } x \in \overline{y} \text{ and } \sigma(x) = X \\ \sigma[x \mapsto Y] & \text{if } x \in \overline{y} \text{ and } x \notin \mathrm{dom}(\sigma) \end{cases}$$

Next, we augment a partial parse tree with a substitution to incrementally accumulate the matches. So a partial tree has the form $\forall \overline{x}.\, A \to^\sigma \alpha.\overline{t}_\beta.\gamma$. We then update four of the deduction rules as shown below, leaving (HYP), (DECL), and (ISLND) unchanged.

$$\text{(PFNSH)}\ \frac{\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^\sigma .\overline{t}_\alpha., i, j]}{\mathcal{G}; H \vdash [\sigma(A) \to \overline{t}_\alpha, i, j]}$$

$$\text{(PIPRED)}\ \frac{\mathcal{G}; H \vdash [t_B, i, j] \quad match(B', B, \sigma_0, \overline{x}) = \sigma \quad \forall \overline{x}.\, A \to \alpha B' \beta \in \mathcal{P} \quad \mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)}{\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^\sigma \alpha.t_B.\beta, i, j]}$$

$$\text{(PRCOMPL)}\ \frac{\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^{\sigma_1} \alpha.\overline{s}_\beta.X'\gamma, i, j] \quad \mathcal{G}; H \vdash [t_X, j, k] \quad match(X', X, \sigma_1, \overline{x}) = \sigma_2}{\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^{\sigma_2} \alpha.\overline{s}_\beta t_X.\gamma, i, k]\}}$$

$$\text{(PLCOMPL)}\ \frac{\mathcal{G}; H \vdash [t_X, i, j] \quad \mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^{\sigma_1} \alpha X'.\overline{s}_\beta.\gamma, j, k] \quad match(X', X, \sigma_1, \overline{x}) = \sigma_2}{\mathcal{G}; H \vdash [\forall \overline{x}.\, A \to^{\sigma_2} \alpha.t_X \overline{s}_\beta.\gamma, i, k]\}}$$

The above rules ensure that we instantiate type parameters in a way that generates the most specific parses for parameterized rules, but there is still the possibility of ambiguities in non-parametric rules. For example, consider the following grammar.

```
Float ::= Int
Float ::= Float "+" Float
Int ::= Int "+" Int
```

The program

```
1 + 2
```

can be parsed at least three different ways, with no coercions from `Int` to `Float`, with two coercions, or with just one coercion. To make sure that our algorithm picks the most specific parse, with no coercions, we make sure to explore derivations in the order of most specific first.

5.3 Grammar Rules with Variable Binders

Variable binding and scoping is an important aspect of programming languages and domain-specific languages are no different in this regard. Consider what would be needed to define the grammar rule to parse a `let` expression such as the following, in which `n` is in scope between the curly brackets.

```
let n = 7 { n * n }
```

To facilitate the definition of binding forms, we add two extensions to our extensible parsing system: labeled symbols [Jim et al. 2010] and a scoping construct [Cardelli et al. 1994]. First, to see an example, consider the below grammar rule.

```
forall T1 T2.
  T2 ::= "let" x:Id "=" T1 "{" x:T1; T2 "}"
```

The identifier `Id` is now labeled with `x`, which provides a way to refer to the string that was parsed as `Id`. The curly brackets are our scoping construct, that is, they are treated specially. The `x:T1` inside the curly brackets declares that `x` is in scope during the parsing of `T2`. Effectively, the grammar is extended with the rule `T1` $\rightarrow$ `x` (but with `T1` replaced by the nonterminal that it is instantiated to, and with `x` replaced by its associated string).

The addition of variable binders and scoping complicates the parsing algorithm because we can no longer proceed purely in a bottom-up fashion. In this example, we cannot parse inside the curly brackets until we have parsed the header of the `let` expression, that is, the variable name and the right-hand side `T1`. Our parsing system handles this by parsing in phases, where initially, all regions of the input enclosed in curly braces are ignored. Once enough of the text surrounding a curly-brace enclosed region has been parsed, then that region is "opened" and the next phase of parsing begins.

5.4 Rule-Action Pairs and Nonterminal-Type Mappings

Sandberg [1982] introduces the notion of a *rule-action pair*, which pairs a grammar rule with a semantic action that provides code to give semantics to the syntax. The following is one of his examples but written using our parsing system on top of Typed Racket.

```
Integer ::= "|" i:Integer "|" => (abs i);
```

The above example defines syntax for the absolute value operation on integers and it defines how to compute the absolute value with code in Typed Racket. After a declaration such as the one above, the programmer can use the notation `|x|` in the rest of the current scope, including subsequent actions within rule-action pairs.

In Sandberg's paper, it seems that rule-action pairs behave like macros. In our system, we provide rule-action pairs that behave like functions as well (with call-by-value semantics). The `=>` operator introduces a function (as in the above example) and the `=` operator introduces a macro. For example, one would want to use a macro to define the syntax of an `if` expression (Figure 7) to avoid always evaluating both branches. We refer to a rule-action pair that defines a function as a *rule-function* and a rule-action pair that defines a macro as a *rule-macro*.

In addition to rule-action pairs, we need a mechanism for connecting nonterminals to types in the host programming language. We accomplish this by simply adding syntax to map a nonterminal to a type. For example, to abbreviate the Typed Racket `Integer` type as `Int`, one would write the following in a grammar.

```
Int = Integer;
```

The implementation of our parsing system translates an input program, containing a mixture of Typed Racket plus our grammar extensions, into a program that is purely Typed Racket. In the following we describe the translation.

A nonterminal-type mapping is translated into a type alias definition. So A `=` T`;` translates to

`(define-type` A T`)`

where T is an expression that evaluates to a type in Typed Racket.

We use two auxiliary functions to compute the arguments of rule-functions and rule-macros for translation. The *support* of a sequence α is the sequence of variables bound in α; the *binders* of α is the sequence of variable bindings in α. In the following definitions we use list comprehension notation.

$$supp(\alpha) = [x_i \mid \alpha_i \in \alpha, \alpha_i = x_i : B_i]$$
$$binders(\alpha) = [x_i : B_i \mid \alpha_i \in \alpha, \alpha_i = x_i : B_i]$$

For both rule-functions and rule-macros, our system generates a unique name f and m, respectively, for use in the Typed Racket output. Then a rule-function of the form $\forall\overline{x}.\, A^f \rightarrow \alpha \Rightarrow e$ is translated to the definition:

`(:` f `(All (`$\overline{x}$`) (`$\overline{B}$ `->` A`)))`
`(define` f `(lambda (`$supp(\alpha)$`)` e`))`

A rule-macro of the form $\forall\overline{x}.\, A^m \rightarrow \alpha = e$ is translated to the following:

`(define-syntax` m
  `(syntax-rules ()`
    `((`m $\overline{x}$ $supp(\alpha)$`)` e`)))`

The type parameters $\overline{x}$ are passed as arguments to macros so they can be used in Typed Racket forms. For example, the rule for `let` expressions in Figure 7 translates to a typed-let expression in Racket using the parameter `T1`.

Next we show the translation of parse trees to Typed Racket, written $[\![t]\!]$. The key idea is that we translate a parse tree for a rule-function pair into a function application, and a parse tree of a rule-macro pair into a macro application,

$$[\![A^f \rightarrow t_\alpha]\!] = (f\ \overline{[\![t_{\alpha'}]\!]})$$
$$[\![\forall\overline{x}.\, A^m \rightarrow^\sigma t_\alpha]\!] = (m\ \sigma(\overline{x})\ \overline{[\![t_{\alpha'}]\!]})$$

where in each case $\alpha' = binders(\alpha)$. Terminals simply translate as themselves, $[\![a]\!] = a$.

5.5 Examples

Here we present examples in which we add syntax for two DSLs to the host language Typed Racket.

5.5.1 Giving ML-like Syntax to Typed Racket

The module in Figure 7 gives ML-like syntax to several operators and forms of the Typed Racket language. The grammar rules for `Int` and `Id` use regular expressions (in Racket syntax) on the right-hand side of the grammar rule.

We then use this module in the following program and save it to the file `let.es`:

```
import ML;
let n = 7 {
  if n < 3 then print 6;
  else print 2 + n * 5 + 5;
}
```

Then we compile it and run the generated `let.rkt` by entering

```
$ esc let.es
$ racket -I typed/racket -t let.rkt -m
42
```

where `esc` is our extensible syntax compiler. The result, of course, is `42`.

5.5.2 A DSL for Sets

The module below defines syntax for converting lists to sets, computing the union and intersection of two sets, and the cardinality of a set. Each rule-macro expands to a Racket library call.

```
module Sets {
  Int = Integer;
  Set = (Setof Int);
  List = (Listof Int);

  Set ::= "{" xs:List "}" = (list->set xs);
  List ::= x:Int = (list x);
  List ::= x:Int "," xs:List = (cons x xs);

  Set ::= s1:Set "|" s2:Set [left,1] =
```

```
module ML {
  // type aliases
  Int = Integer;
  Bool = Boolean;

  // functions
  Int ::= x:Int "+" y:Int [left,1] => (+ x y);
  Int ::= x:Int "*" y:Int [left,2] => (* x y);
  Bool ::= x:Int "<" y:Int => (< x y);
  forall T.
    Void ::= "print" x:T ";" => (displayln x);

  // macros
  forall T.
    T ::= "if" t:Bool "then" e1:T "else" e2:T =
      (if t e1 e2);
  forall T1 T2.
    T2 ::= "let" x:Id "=" y:T1 "{" x:T1; z:T2 "}" =
      (let: ([x : T1 y]) z);
  forall T1 T2.
    T2 ::= e1:T1 e2:T2 [left] =
      (begin e1 e2);

  // tokens
  Int ::= #rx"^[0-9]+$";
  Id ::= #rx"^[a-zA-Z][a-zA-Z0-9]*$";
}
```

Figure 7. An example of giving ML-like syntax to Typed Racket.

```
      (set-union s1 s2);
    Set ::= s1:Set "&" s2:Set [left,2] =
      (set-intersect s1 s2);
    Int ::= "|" s:Set "|" = (set-count s);
  }
```

After importing this DSL, programmers can use the set syntax directly in Typed Racket. We can also combine the `Sets` module with the `ML` module from before, for example:

```
import ML, Sets;
let A = {1, 2, 3} {
  let B = {2, 3, 4} {
    let C = {3, 4, 5} {
      print |A & C|;
      print A | B & C;
    }
  }
}
```

Saving this program in `sets.es`, we can then compile and run it:

```
$ esc sets.es
$ racket -I typed/racket -t sets.rkt -m
1
#<set: 1 2 3 4>
```

6. Related Work

There have been numerous approaches to extensible syntax for programming languages. In this section, we summarize the approaches and discuss how they relate to our work. We organize this discussion in a roughly chronological order.

In the Lithe language, Sandberg [1982] merges the notion of grammar rule and macro definition and integrates parsing and type checking. Unfortunately, he does not describe his parsing algorithm. Aasa et al. [1988] augments the ML language with extensible syntax for dealing with algebraic data types. They develop a generalization of the Earley algorithm that performs Hindley-Milner type inference during parsing. However, Pettersson and Fritzson [1992] report that the algorithm was too inefficient to be practically usable. Pettersson and Fritzson [1992] build a more efficient system based on LR(1) parsing. Of course, LR(1) parsing is not suitable for our purposes because LR(1) is not closed under union, which we need to compose DSLs. Several later works also integrate type inference into the Earley algorithm [Missura 1997, Wieland 2009]. It may be possible to adapt these ideas to enable our approach to handle languages with type inference.

Cardelli et al. [1994] develop a system with extensible syntax and lexical scoping. That is, their system supports syntax extensions that introduce variable binders. Their work inspires our treatment of variable binders in Section 5.3. Cardelli et al. [1994] base their algorithm on LL(1) parsing, which is also not closed under union. Also, their system differs from ours in that parsing and type checking are separate phases. The OCaml language comes with a preprocessor, Camlp4, that provides extensible syntax [de Rauglaudre 2002]. The parsing algorithm in Camlp4 is "something close to LL(1)".

Goguen et al. [1992] provide extensible syntax in the OBJ3 language in the form of mixfix operators. In OBJ3, types (or sorts) play some role in disambiguation, but their papers do not describe the parsing algorithm. There is more literature regarding Maude [Clavel et al. 1999], one of the descendents of OBJ3. Maude uses the SCP algorithm of Quesada [1998], which is bottom-up and bidirectional, much like our island parser. However, we have not been able to find a paper that describes how types are used for disambiguation in the Maude parser.

The Isabelle Proof Assistant [Paulson 1994] provides support for mixfix syntax definitions. The algorithm is a variant of chart parsing and can parse arbitrary CFGs, including ambiguous ones. When there is ambiguity, a parse forest is generated and then a later type checking pass (based on Hindley-Milner type inference) prunes out the ill-typed trees.

Ranta [2004] develops the Grammatical Framework which integrates context free grammars with a logical framework based on type theory, that is, a rich type system with dependent types. His framework handles grammar rules with variable binders by use of higher-order abstract syntax. The implementation uses the Earley algorithm and type checks after parsing, similar to Isabelle.

Several systems use Ford's Parsing Expression Grammar (PEG) formalism [Ford 2004]. PEGs are stylistically similar to CFGs; however, PEGs avoid ambiguity by introducing a prioritized choice operator for rule alternatives and PEGs disallow left-recursive rules. We claim that these two restrictions are not appropriate for composing DSLs. The order in which DSLs are imported should not matter and DSL authors should be free to use left recursion if that is the most natural way to express their grammar.

Danielsson and Norell [2008] investigate support for mixfix operators for Agda using parser combinators with memoization, which is roughly equivalent to the Earley algorithm. Their algorithm does not use type-based disambiguation during parsing, but they note that a type-checking post-processor could be used to filter parse trees, as is done in Isabelle.

The MetaBorg [Bravenboer et al. 2005] system provides extensible syntax in support of embedding DSLs in general purpose languages. MetaBorg is built on the Stratego/XT toolset which in turn used the syntax definition framework SDF [Heering et al. 1989] which uses scannerless GLR to parse arbitrary CFGs. Like Isabelle, the MetaBorg system performs type-based disambiguation to prune ill-typed parse trees from the resulting parse forest. Our treatment precedence and associativity is based on their notion of disambiguation filter [van den Brand et al. 2002]. We plan to explore the

scannerless approach in the future. Bravenboer and Visser [2009] look into the problem of composing DSLs and investigate methods for composing parse tables. We currently do not create parse tables, but we may use these ideas in the future to further optimize the efficiency of our algorithm.

Jim et al. [2010] develop a grammar formalism and parsing algorithm to handle data-dependent grammars. One of the contributions of their work is ability to bind parsing results to variables that can then be used to control parsing. We use this idea in Section 5.3 to enable grammar rules with variable binding. Their algorithm is a variation of the Earley algorithm and does not perform type-based disambiguation but it does provide attribute-directed parsing.

7. Conclusions

In this paper we presented a new parsing algorithm, *type-oriented island parsing*, that is the first parsing algorithm to be constant time with respect to the size of the grammar under the assumption that the grammar is sparse. (Most parsing algorithms are linear with respect to the size of the grammar.) Our motivation for developing this algorithm comes from the desire to compose domain-specific languages, that is, to simultaneously import many DSLs into a software application.

We present an extensible parsing system that provides a front-end to a host language, such as Typed Racket, enabling the definition of macros and functions together with grammar rules that provide syntactic sugar. Our parsing system provides precedence and associativity annotations, parameterized grammar rules, and grammar rules with variable binders and scope.

In the future we plan to extend the syntax of nonterminals to represent structural types, formally prove the correctness of type-oriented island parsing, pursue further opportunities to improve the performance of the algorithm, and provide diagnostics for helping programmers resolve the remaining ambiguities that are not addressed by typed-based disambiguation.

References

A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *ACM Conference on LISP and Functional Programming*, LFP, pages 96–105. ACM, 1988. doi: 10.1145/62678.62688.

J. Allen. *Natural language understanding (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1995.

D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Fourth Annual USENIX Tcl/Tk Workshop*, 1996.

M. Bravenboer and E. Visser. Software language engineering. chapter Parse Table Composition, pages 74–94. Springer-Verlag, Berlin, Heidelberg, 2009. doi: http://dx.doi.org/10.1007/978-3-642-00434-6_6.

M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–172. Springer-Verlag, 2005.

L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, DEC SRC, 2 1994.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Technical report, 1999.

N. A. Danielsson and U. Norell. Parsing mixfix operators. In *20th International Symposium on the Implementation and Application of Functional Languages*, 2008.

D. de Rauglaudre. *Camlp4 reference manual*. INRIA, 2002.

J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13: 94–102, 1970. doi: 10.1145/362007.362035.

J. C. Earley. *An efficient context-free parsing algorithm*. PhD thesis, Pittsburgh, PA, USA, 1968.

B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 111–122. ACM, 2004. doi: 10.1145/964001.964011.

J. A. Goguen, T. Winker, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Applications of Algebraic Specification using OBJ*. Cambridge University Press, 1992.

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75, 1989. doi: 10.1145/71605.71607.

T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 417–430, New York, NY, USA, 2010. ACM.

D. Jurafsky and J. Martin. *Speech and Language Processing*. Pearson Prentice Hall, 2009.

L. C. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 918–932, New York, NY, USA, 2010. ACM. doi: http://doi.acm.org/10.1145/1869459.1869535.

M. Kay. *Algorithm schemata and data structures in syntactic processing*, pages 35–70. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

S. Missura. *Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing*. PhD thesis, ETH Zurich, 1997.

L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

M. Pettersson and P. Fritzson. A general and practical approach to concrete syntax objects within ml. In *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

J. F. Quesada. The scp parsing algorithm : computational framework and formal properties. In *Procesamiento del lenguaje natural*, number 23, 1998.

A. Ranta. Grammatical framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

D. Sandberg. Lithe: a language combining a flexible syntax and classes. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 142–145, New York, NY, USA, 1982. ACM. doi: 10.1145/582153.582169.

K. Sikkel. Parsing schemata and correctness of parsing algorithms. *Theoretical Computer Science*, 199(1-2):87–103, 1998. doi: 10.1016/S0304-3975(97)00269-7.

O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Conference on Computational Linguistics*, COLING, pages 636–641. Association for Computational Linguistics, 1988.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. doi: 10.1145/1328438.1328486.

M. Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2*, pages 756–764, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *International Conference on Compiler Construction*, CC, pages 143–158. Springer-Verlag, 2002.

E. Visser. A case study in optimizing parsing schemata by disambiguation filters. In *International Workshop on Parsing Technology (IWPT'97)*, pages 210–224, Boston, USA, September 1997. Massachusetts Institute of Technology.

J. Wieland. *Parsing Mixfix Expressions*. PhD thesis, Technische Universitat Berlin, 2009.